Announcements

- HW2 and WQ2 released
  - Both due next Tuesday

- Please fill in the Azure questionnaire by tonight!
  - See HW2 writeup for details

Simple Aggregations

Five basic aggregate operations in SQL

- `SELECT count(*) FROM Purchase`
- `SELECT sum(quantity) FROM Purchase`
- `SELECT avg(price) FROM Purchase`
- `SELECT min(quantity) FROM Purchase`
- `SELECT max(quantity) FROM Purchase`

Except `count`, all aggregations apply to a single attribute.

Semantics of SQL With Group-By

```
SELECT S
FROM R1, ..., Rn
WHERE C1
GROUP BY a1, ..., ak
HAVING C2
```

Evaluation steps:
1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes `a1, ..., ak`
3. Apply condition `C2` to each group (may have aggregates)
4. Compute aggregates in `S` and return the result

Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as "TotalSold"
Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as "TotalSold"

```sql
FROM Purchase
GROUP BY month
HAVING sum(quantity) < 10
```

WHERE vs HAVING

- WHERE condition is applied to individual rows
  - The rows may or may not contribute to the aggregate
  - No aggregates allowed here

- HAVING condition is applied to the entire group
  - Only applicable if GROUP BY is involved
  - Entire group is returned, or removed
  - May use aggregate functions on the group
Aggregate + Join
For each manufacturer, compute how many products with price > $100 they sold

Problem: manufacturer is in Product, price is in Purchase...

--- step 1: think about their join
SELECT ...
FROM Product x, Purchase y
WHERE x.pid = y.product_id
    and y.price > 100

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>150</td>
</tr>
<tr>
<td>Canon</td>
<td>300</td>
</tr>
<tr>
<td>Hitachi</td>
<td>180</td>
</tr>
</tbody>
</table>

--- step 2: do the group-by on the join
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.pid = y.product_id
    and y.price > 100
GROUP BY x.manufacturer

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>2</td>
</tr>
<tr>
<td>Canon</td>
<td>1</td>
</tr>
</tbody>
</table>

Variant:
For each manufacturer, compute how many products with price > $100 they sold in each month

SELECT x.manufacturer, y.month, count(*)
FROM Product x, Purchase y
WHERE x.pid = y.product_id
    and y.price > 100
GROUP BY x.manufacturer, y.month

Including Empty Groups
- In the result of a group by query, there is one row per group in the result

SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.pname = y.product
GROUP BY x.manufacturer

Count(*) is never 0
Including Empty Groups

```
SELECT x.manufacturer, count(y.pid)
FROM Product x LEFT OUTER JOIN Purchase y
ON x.pname = y.product
GROUP BY x.manufacturer
```

Count(pid) is 0 when all pid's in the group are NULL

What we have in our SQL toolbox

- Projections (SELECT * / SELECT c1, c2, ...)
- Selections (aka filtering) (WHERE cond)
- Joins (inner and outer)
- Aggregates
- Group by
- Inserts, updates, and deletes

Make sure you read the textbook!

Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries
- A subquery may occur in:
  - A SELECT clause
  - A FROM clause
  - A WHERE clause

- Rule of thumb: avoid nested queries when possible
  - But sometimes it's impossible, as we will see

Subqueries...

- Can return a single value to be included in a SELECT clause
- Can return a relation to be included in the FROM clause, aliased using a tuple variable
- Can return a single value to be compared with another value in a WHERE clause
- Can return a relation to be used in the WHERE or HAVING clause under an existential quantifier

1. Subqueries in SELECT

```
Product (pname, price, cid)
Company (cid, cname, city)
```

For each product return the city where it is manufactured

```
SELECT x.pname, (SELECT Y.city
FROM Company Y
WHERE Y.cid= X.cid) as City
FROM Product X
```

What happens if the subquery returns more than one city?

We get a runtime error (and SQLite simply ignores the extra values...)

Whenever possible, don't use a nested queries:

```
SELECT X.pname, (SELECT Y.city
FROM Company Y
WHERE Y.cid= X.cid) as City
FROM Product X
```

We have "unnested" the query
1. Subqueries in SELECT

Compute the number of products made by each company

```sql
SELECT DISTINCT C.cname, (SELECT count(*) FROM Product P WHERE P.cid=C.cid)
FROM Company C
```

Better: we can unnest using a `GROUP BY`

```sql
SELECT C.cname, count(*)
FROM Company C, Product P
WHERE C.cid=P.cid
GROUP BY C.cname
```

But are these really equivalent?

```sql
SELECT DISTINCT C.cname, (SELECT count(*) FROM Product P WHERE P.cid=C.cid)
FROM Company C
SELECT C.cname, count(*)
FROM Company C, Product P
WHERE C.cid=P.cid
GROUP BY C.cname
```

No! Different results if a company has no products

```sql
SELECT C.cname, count(pname)
FROM Company C LEFT OUTER JOIN Product P ON C.cid=P.cid
GROUP BY C.cname
```

2. Subqueries in FROM

Find all products whose prices is > 20 and < 500

```sql
SELECT X.pname
FROM (SELECT * 
      FROM Product P AS Y 
      WHERE price > 20) as X 
WHERE X.price < 500
```

Try unnest this query!

```sql
SELECT X.pname 
FROM (SELECT * 
      FROM Product P AS Y 
      WHERE price > 20) as X 
WHERE X.price < 500
```
2. Subqueries in FROM

Find all products whose prices is > 20 and < 500

```
SELECT X.pname
FROM (SELECT *
     FROM Product AS Y
     WHERE price > 20) as X
WHERE X.price < 500
```

Try unnest this query!

Side note: This is not a correlated subquery. (why?)

2. Subqueries in FROM

Sometimes we need to compute an intermediate table only to use it later in a SELECT-FROM-WHERE

- Option 1: use a subquery in the FROM clause
- Option 2: use the WITH clause
  – See textbook for details